HELIX-RC: An Architecture-Compiler Co-Design for Automatic Parallelization of Irregular Programs

Simone Campanoni Kevin Brownell Svilen Kanev Timothy M. Jones[+] Gu-Yeon Wei David Brooks

Harvard University
Cambridge, MA, USA

University of Cambridge[+]
Cambridge, UK

Abstract

Data dependences in sequential programs limit parallelization because extracted threads cannot run independently. Although thread-level speculation can avoid the need for precise dependence analysis, communication overheads required to synchronize actual dependences counteract the benefits of parallelization. To address these challenges, we propose a lightweight architectural enhancement co-designed with a parallelizing compiler, which together can decouple communication from thread execution. Simulations of these approaches, applied to a processor with 16 Intel Atom-like cores, show an average of 6.85× performance speedup for six SPEC CINT2000 benchmarks.

1. Introduction

In today's multicore era, program performance largely depends on the amount of thread level parallelism (TLP) available. While some computing problems often translate to either inherently parallel or easy-to-parallelize numerical programs, sequentially designed, non-numerical programs with complicated control (e.g., execution path) and data flow (e.g., aliasing) are much more common, but difficult to analyze precisely. These non-numerical programs are the focus of this paper. While conventional wisdom is that non-numerical programs cannot make good use of multiple cores, research in the last decade has made steady progress towards extracting TLP from complex, sequentially-designed programs such as the integer benchmarks from SPEC CPU suites [3, 6, 27, 45, 48]. To further extend this body of research, this paper presents lightweight architectural enhancements for fast inter-core communication in order to support advances in a custom compiler framework that parallelizes loop iterations across multiple cores within a chip multiprocessor.

Performance gains sought by parallelizing loop iterations of non-numerical programs depend on two key factors: (i) accuracy of the data dependence analysis and (ii) speed of communication provided by the underlying computer architecture to satisfy the dependences. Unfortunately, complex control and data flow in non-numerical programs—both exacerbated by ambiguous pointers and ambiguous indirect calls—make accurate data dependence analysis difficult. In addition to *actual* dependences that require communication between cores, a compiler must conservatively handle *apparent* dependences never realized at runtime. While thread level speculation (TLS) avoids the need for accurate data dependence analysis by speculating that some apparent dependences are not realized [29, 38, 39], TLS suffers overheads to support misspeculation and must therefore target relatively large loops to amortize penalties.

In contrast to existing parallelization solutions, we propose an alternate strategy that instead targets *small loops*, which are much easier to analyze via state-of-the-art control and data flow analysis, significantly improving accuracy. Furthermore, this ease of analysis enables transformations that simply re-compute shared variables in order to remove a large fraction of actual dependences. This strategy increases TLP and reduces core-to-core communication. Such optimizations do not readily translate to TLS because the complexity of TLS-targeted code typically spans multiple procedures in larger loops. Finally, our data shows parallelizing small hot loops yields high program coverage and produces meaningful speedups for the non-numerical programs in the SPEC CPU2000 suite.

Targeting small loops presents its own set of challenges. Even after extensive code analysis and optimizations, small hot loops will retain actual dependences, typically to share dynamically allocated data. Moreover, since loop iterations of small loops tend to be short in duration (less than 25 clock cycles on average), they require frequent, memory-mediated communication. Attempting to run these iterations in parallel demands low-latency core-to-core communication for memory traffic, something not available in commodity multicore processors.

To meet these demands, we present HELIX-RC, a co-designed architecture-compiler parallelization framework for chip multiprocessors. The compiler identifies what data must be shared between cores and the architecture proactively circulates this data along with synchronization signals among cores. Rather than waiting for a request, this proactive communication immediately circulates shared data, as early as possible—decoupling communication from computation. HELIX-RC builds on the HCCv1 compiler, developed for the first iteration of HELIX [6, 7], that automatically generates parallel code for commodity multicore processors. Because performance improvements from HCCv1 saturate at four cores due to communication latency, we propose *ring cache*—an architectural enhancement that facilitates low-latency core-to-core communication—to satisfy inter-thread memory dependences and relies on guarantees provided by the co-designed *HCCv3* compiler to keep it lightweight.

HELIX-RC automatically parallelizes non-numerical program with unmatched performance improvements. Across a range of SPEC CINT2000 benchmarks, decoupling communication enables a three-fold improvement in performance when compared

978-1-4799-4394-4/14/$31.00

to HCCv1, on a simulated multicore processor consisting of 16 Atom-like, *in-order* cores with a ring cache with 1KB per node of memory (32× smaller than the L1 data cache). The proposed system offers an average speedup of 6.85× when compared to running un-parallelized code on a single core. Detailed evaluations show that even with a conservative ring cache configuration, HELIX-RC is able to achieve 95% of the speedup possible with unlimited resources (i.e., unbounded bandwidth, instantaneous inter-core communication, and unconstrained size). Moreover, simulations for a HELIX-RC system comprising 16 *out-of-order* cores show 3.8× performance speedup for the same set of non-numerical programs. This result confirms HELIX-RC's ability to extract TLP on top of the instruction level parallelism (ILP) provided by an out-of-order processor.

The remainder of this paper further describes the motivation for and results of implementing HELIX-RC. We first review the limitations of compiler-only improvements and identify co-design opportunities to improve TLP of loop iterations. Next, we explore the speedups obtained by decoupling communication from computation with compiler support. After describing the overall HELIX-RC approach, we delve deeper into both the compiler and the hardware enhancement. Finally, we use a detailed simulation framework to evaluate the performance of HELIX-RC and analyze its sensitivity to architectural parameters.

2. Background and Opportunities

2.1. Limits of compiler-only improvements

To understand what limits the performance of parallel code extracted from non-numerical programs, we started with HCCv1 [6, 7], a state-of-the-art parallelizing compiler.

HCCv1. This first generation compiler automatically generates parallel threads from sequential programs by distributing successive loop iterations across adjacent cores within a single multicore processor, similar to conventional DOACROSS parallelism [10]. Since there are data dependences between loop iterations (i.e., *loop-carried dependences*), some segments of a loop's body—called *sequential segments*—must execute in iteration order on the separate cores to preserve the semantics of sequential code. Synchronization operations mark the beginning and end of each sequential segment.

HCCv1 includes a large set of code optimizations (e.g., code scheduling, method inlining, loop unrolling), most of which are specifically tuned to extract TLP. Despite this, performance improvements obtained by the original HCCv1 compiler saturate at four cores, due to high core-to-core communication latency.

HCCv2. We first improved code analysis and transformation. Specifically, we increased the accuracy of both data dependence and induction variable analysis, and we added other transformations to extract more parallelism (e.g., scalar expansion, scalar renaming, parallel reductions, and loop splitting [1]). We call this improved compiler HCCv2.

Figure 1 compares speedups for HCCv1 and HCCv2 based on simulations of parallel code generated by each when targeting a 16-core processor with an optimistic 10-cycle core-to-core communication latency.[1] The engineering improvements of HCCv2 significantly raised speedups for numerical programs (SPEC CFP2000) over HCCv1 from 2.4× to 11×. HCCv2 successfully parallelized the numerical programs because the accuracy of the data dependence analysis is high for loops at almost any level of the loop nesting hierarchy. Furthermore, the improved compiler removed the remaining actual dependences among registers (e.g., via parallel reduction) to generate loops with long iterations that can run in parallel on different cores.

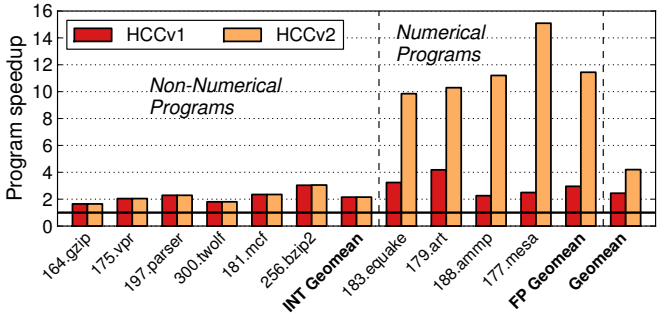


Figure 1: Improving the HCCv1 compiler alone does not improve performance for SPEC CINT2000 benchmarks.

Unfortunately, non-numerical programs (SPEC CINT) are not as compliant to compiler improvements and saw little to no benefit from HCCv2. Because core-to-core communication in conventional systems is expensive, the compiler must parallelize large loops (the larger the loop with loop-carried dependences, the less frequently cores synchronize), which limits the accuracy of dependence analysis and thereby limits TLP extraction. This is why HELIX-RC focuses on small (hot) loops to parallelize this class of programs. Our hypothesis is that modest architectural enhancements, co-designed with a compiler that targets small loops, can successfully parallelize non-numerical programs.

2.2. Opportunity

There is an opportunity to aggressively parallelize non-numerical programs based on the following insights: (i) small loops are easier to analyze with high accuracy; (ii) predictable computation means most of the required communication updates shared memory locations; (iii) we can efficiently satisfy communication demands of actual dependences for small loops with low-latency, core-to-core communication; and (iv) proactive communication efficiently hides communication latencies.

Accurate data dependence analysis is possible for small loops in non-numerical programs. The accuracy of data dependence analysis increases for smaller loops because (i) there is less code—therefore less complexity—to analyze and (ii) the number of possible aliases of a pointer in the code scales down with code size. In other words, we can avoid conservative pointer aliasing assumptions that lower accuracy for large loops.

To evaluate the accuracy of data dependence analysis for small loops using modern compilers, we started with a state-of-the-art

[1] Details of this experiment are in Section 6.

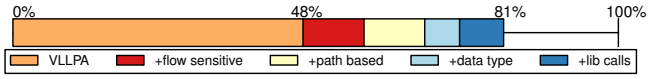


Figure 2: Accuracy of data dependence analysis for small hot loops in SPEC CINT2000 benchmarks.

Figure 3: Predictability of variables reduces register communication.

analysis called VLLPA [13]. Figure 2 shows the starting accuracy (i.e., average number of actual data dependences compared to all dependences identified for our set of loops) of this analysis is 48%. To improve accuracy, we extended VLLPA (i) to be fully flow sensitive [8], which tracks values of both registers and memory locations according to their position in the code; (ii) to be path-based, which names runtime locations by how they are accessed from program variables [11]; (iii) to exploit data type and type casting information to conservatively eliminate incompatible aliases; and (iv) to exploit standard library call semantics. Figure 2 shows that these extensions increase accuracy for small loops to 81%. As a result, most of the loop-carried data dependences identified by the compiler are actual and therefore require core-to-core communication.

Most required communication is to update shared memory locations. Sharing data among loop iterations requires core-to-core communication to propagate new values when loop iterations run on different cores. However, if new values are predictable (e.g., incrementing a shared variable at every iteration), communication can be avoided. We extended the variable analysis in HCCv1 to capture the following predictable variables: (i) induction variables where the update function is a polynomial up to the second order; (ii) accumulative, maximum, and minimum variables; (iii) variables set but not used until after the loop; and (iv) variables set in every iteration even when the updated value is not constant.[2] If a variable falls into any of these categories, each core can re-compute its correct value independently.

Exploiting the predictability of variables, again for small loops in non-numerical programs, allows the compiler to remove a large fraction of the communication required to share registers. Figure 3 compares a naive solution that propagates new values for all loop-carried data dependences (100%) versus a solution that exploits variable predictability. By re-computing variables, the majority of the remaining communication is to share memory locations rather than registers.

Communication for small hot loops must be fast. While the simplicity of small loops allows for easy analysis, small loops have short iterations—typically less than 100 clock cycles long. However, because these short iterations require (at least) some communication to run in parallel, efficient parallel execution demands a low-latency core-to-core communication mechanism.

To better understand this need for fast communication, Figure 4a plots a cumulative distribution of average iteration execution times on a single Atom-like core (described in Section 6) for the set of hot loops from SPEC CINT2000 benchmarks chosen for parallelization by HELIX-RC. The shaded portion of the plot shows that more than half of the loop iterations complete within 25 clock cycles. The plot also delineates the measured core-to-core round trip communication latencies for three modern multicore processors. Even for the shortest-latency machine, Ivy Bridge, 75 cycles is much too long for the majority of these short loops. Of course, a conventional region-extending transformation such as loop unrolling could lengthen the duration of these inner loops, but this would also increase the lengths of sequential segments, reducing exploitable parallelism.

Proactive communication achieves low latency by decoupling communication from computation. A compiler must conservatively assume dependences exist between all iterations for most of loop-carried dependences in non-numerical programs. Because of the complexity of control and data flow in such programs, a compiler cannot easily infer the distance between a loop iteration that generates data and the ones that consume it. For conventional synchronization approaches [6, 25, 26, 43, 47, 48], this assumption of dependences between all subsequent iterations leads to *sequential chain*s that severely limit the performance sought by running loop iterations in parallel.[3]

These sequential chains, which include both communication and computation, have two sources of inefficiency. First, adjacent-core synchronization often turns out to *not* be necessary for every link of these chains. Second, when data forwarding is initiated lazily (at request time), it blocks computation while waiting for data transfers between cores.

Finally, for loops parallelized by HELIX-RC, most communication is not between successive loop iterations. Hence, because HELIX-RC distributes successive loop iterations to adjacent cores, most communication is not between adjacent cores. Figure 4b charts the distribution of undirected distances between data-producing cores and the first consumer core on a platform with 16 cores organized in a ring. Only 15% of those transfers are between adjacent cores. Moreover, Figure 4c shows that most of the shared values (86%) from these loops are consumed by multiple cores. Since consumers of shared values are not known at compile time, HELIX-RC implements a mechanism that proactively broadcasts data and signals to all other cores. Such proactive communication, which does not block computation, is the cornerstone of the HELIX-RC approach.

3. The HELIX-RC Solution

The goal of HELIX-RC is to decouple *all* communication required to efficiently run iterations of small hot loops in parallel. This is realized by *decoupling value forwarding from value generation* and by *decoupling signal transmission from synchronization*. We now show how HELIX-RC achieves such decoupling.

[2] This is an example of code replication. Details about this transformation are outside the scope of this paper.

[3] Others have called this chain a *critical forwarding path* [40, 48].

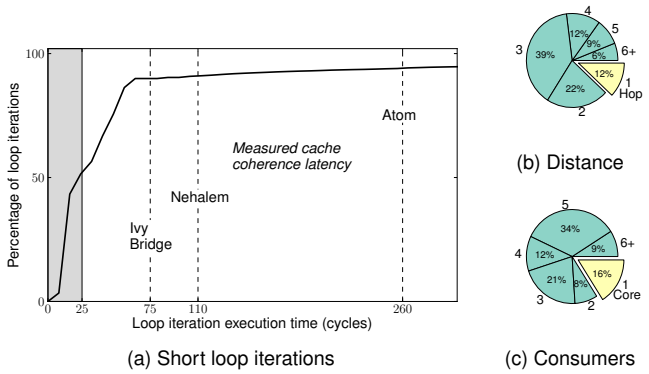


Figure 4: Small hot loops have short iterations that send data over multiple hops and to multiple cores.

3.1. Approach

HELIX-RC is a co-design of its compiler (*HCCv3*) and architectural enhancements (the *ring cache*). HCCv3 distinguishes parallel code (i.e., code outside any sequential segment) from sequential code (i.e., code within sequential segments) by using two instructions that extend the instruction set. The ring cache is a ring network that connects together *ring nodes* attached to each core in the processor to operate during sequential segments as a distributed first-level cache that precedes the private L1 cache. Because it relies on compiler-guaranteed properties of the code, the hardware support can be simple and efficient. The next paragraphs summarize the main components of HELIX-RC.

ISA. We introduce a pair of instructions—`wait` and `signal`—that mark the beginning and end of a sequential segment. Each of these instructions has an integer value as a parameter that identifies the particular sequential segment. The `wait` instruction blocks execution of the core that issued it (e.g., `wait 3`) until all other cores have finished executing the corresponding sequential segment, which they signify by executing the appropriate `signal` instruction (e.g., `signal 3`).

Compiler. HCCv3 takes sequential programs and parallelizes loops that are most likely to speed up performance when their iterations execute in parallel. Only one loop runs in parallel at a time and its successive iterations run on cores organized as a unidirectional ring.

To satisfy loop-carried data dependences, HCCv3 keeps the execution of sequential segments in iteration order by inserting `wait` and `signal` instructions to delimit the entry and exit points of these segments. In this way, HCCv3 guarantees that accesses to a variable or another memory location that might need to be shared between cores are always within sequential segments. Moreover, shared variables (normally allocated to registers in sequential code) are mapped to specially-allocated memory locations. Hence, their accesses within sequential segments occur via memory operations.

Core. A core forwards all memory accesses *within* sequential segments to its local ring node. All other memory accesses (not within a sequential segment) go through the private L1 cache. To determine whether the executing code is part of a sequential segment or not, a core simply counts the number of executed `wait` and `signal` instructions. If more `wait`s have been executed than matching `signal`s, then the executing code belongs to a sequential segment.

Memory. The ring cache is a connected ring of nodes, one per core. Each ring node has a cache array that satisfies both loads and stores received from its attached core. HELIX-RC does not require other changes to the existing memory hierarchy because the ring cache orchestrates interactions with it. To avoid any changes to conventional cache coherence protocols, the ring cache permanently maps each memory address to a unique ring node. All accesses from the distributed ring cache to the next cache level (L1) go through the associated node for a corresponding address.

3.2. Decoupling communication from computation

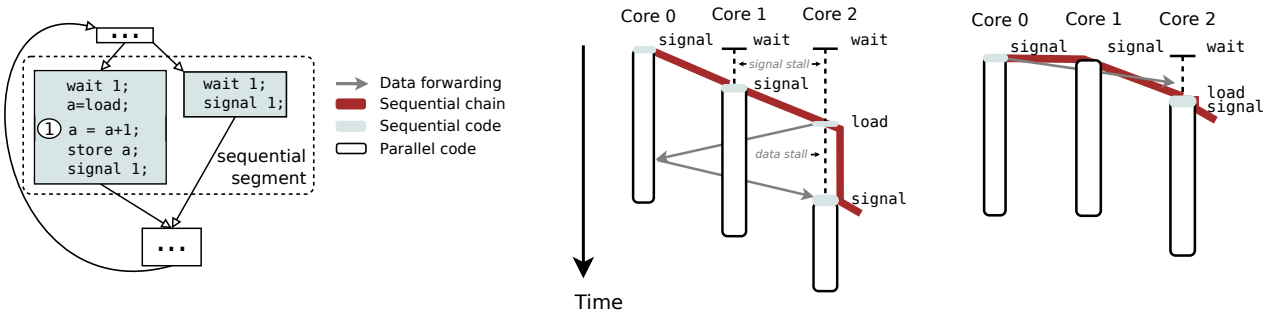
Having seen the main components of HELIX-RC, we describe how they interact to efficiently decouple communication from computation.

Shared data communication. HELIX-RC decouples communication of variables and other shared data locations by propagating new shared data through the ring cache as soon as it is generated. Once a ring node receives a store, it records the new value and proactively forwards its address and value to an adjacent node in the ring cache, all without interrupting the execution of the attached core. The value then propagates from node to node through the rest of the ring without interrupting the computation of any core—*decoupling communication from computation*.

Synchronization. Given the difficulty of determining which iteration depends on which in non-numerical programs, compilers typically make the conservative assumption that an iteration depends on all of its predecessor iterations. Therefore, a core cannot execute sequential code until it is unblocked by its predecessor [6, 25, 40]. Moreover, an iteration unblocks its successor only if both it and its predecessors have executed this sequential segment or if they are not going to. This execution model leads to a chain of signal propagation across loop iterations that includes unnecessary synchronization: even if an iteration is not going to execute sequential code, it still needs to synchronize with its predecessor before unblocking its successor.

HELIX-RC removes these synchronization overheads by enabling an iteration to detect the readiness of all predecessor iterations, not just one. Therefore, once an iteration forgoes executing the sequential segment, it immediately notifies its successor without waiting for its predecessor. Unfortunately, while HELIX-RC removes unnecessary synchronization, it increases the number of signals that can be in flight simultaneously.

HELIX-RC relies on the new `signal` instruction to handle synchronization signals efficiently. Synchronization between a producer and a consumer includes (i) the producer generating a signal, (ii) the consumer requesting that signal, and (iii) signal transmission between the two. On a conventional multicore, which relies on a pull-based memory hierarchy for communication, signal transmission is inherently lazy, and signal request and



(a) Parallel code (b) Coupled communication (c) Decoupled communication

Figure 5: Example illustrating benefits of decoupling communication from computation.

transmission get serialized. On the other hand, in HELIX-RC, `signal` instructs the ring cache to proactively forward a signal to all other nodes in the ring without interrupting any of the cores, thereby *decoupling signal transmission from synchronization*.

Code example. Given the importance of these decoupling mechanisms to fully realize performance benefits, let's explore how HELIX-RC implements them using a concrete example. The code in Figure 5(a), abstracted for clarity, represents a small hot loop from 175.vpr of SPEC CINT2000 that is responsible for 55% of the total execution time of that program. It contains a sequential segment with two possible execution paths. The left path contains an actual dependence where instances of instruction `1` in an iteration use values from previous iterations. The right path does not depend on prior data. Because the compiler cannot predict the execution path of a particular iteration (due to complex control flow), it must assume that instruction `1`, in any given iteration, depends on the previous iteration. Therefore, it must synchronize all successive iterations by inserting `wait` and `signal` instructions on every execution path. Figure 5(b) highlights this sequential chain in red. Now, assume only iterations 0 and 2, running on cores 0 and 2, respectively, execute instruction `1`. Then, this sequential chain is unnecessarily long because of the superfluous `wait` in iteration 1. Each iteration waits (via the `wait` instruction) for the signal generated by the `signal` instruction of the previous iteration. Also, iterations that update `a` (iterations 0 and 2) must load previous values first (using a regular load). Hence, two sets of stalls slow down the chain. First, iteration 1 performs unnecessary synchronization (signal stalls), because it only contains parallel code. Second, lazy forwarding of the shared data leads to data stalls, because the transfer only begins when requested, at a load, rather than when generated, at a store.

HELIX-RC proactively communicates data and synchronization signals between cores, which leads to the more efficient scenario shown in Figure 5(c). The sequential chain now only includes the delay required to satisfy the dependence—communication updating a shared value.

4. Compiler

The decoupled execution model of HELIX-RC described so far is possible given the tight co-design of the compiler and architecture. In this section, we focus on compiler-guaranteed code properties that enable a lightweight ring cache design, and follow up with code optimizations that make use of the ring cache.

Code properties.

- Only one loop can run in parallel at a time. Apart from a dedicated core responsible for executing code outside parallel loops, each core is either executing an iteration of the current loop or waiting for the start of the next one.
- Successive loop iterations are distributed to threads in a round-robin manner. Since each thread is pinned to a predefined core, and cores are organized in a unidirectional ring, successive iterations form a logical ring.
- Communication between cores executing a parallelized loop occurs only within sequential segments.
- Different sequential segments always access different shared data. HCCv3 only generates multiple sequential segments when there is no intersection of shared data. Consequently, instances of distinct sequential segments may run in parallel.
- At most two signals per sequential segment emitted by a given core can be in flight at any time. Hence, only two signals per segment need to be tracked by the ring cache.

This last property eliminates unnecessary `wait` instructions while keeping the architectural enhancement simple. Eliminating `wait`s allows a core to execute a later loop iteration than its successor (significantly boosting parallelism). Future iterations, however, produce signals that must be buffered. The last code property prevents a core from getting more than one "lap" ahead of its successor. So when buffering signals, each ring cache node only needs to recognize two types—those from the past and those from the future.

Code optimizations. In addition to the optimizations of HCCv2, HCCv3 includes ones that are essential for best performance of non-numerical programs on a ring-cache-enhanced architecture: aggressive splitting of sequential segments into smaller code blocks; identification and selection of small hot loops; and elimination of unnecessary `wait` instructions.

Sizing sequential segments poses a tradeoff. Additional segments created by splitting run in parallel with others, but extra segments entail extra synchronization, which adds communication overhead. Thanks to decoupling, HCCv3 can split more

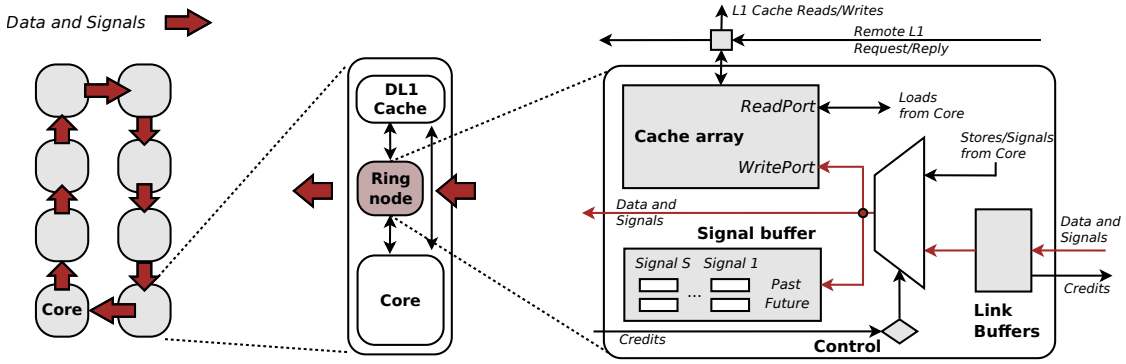


Figure 6: Ring cache architecture overview. From left to right: overall system; single core slice; ring node internal structure.

aggressively than HCCv2 to significantly increase TLP. Note that segments cannot be split indefinitely—each shared location must belong to only one segment.

To identify small hot loops that are most likely to speed up when their iterations run in parallel, HCCv3 includes a profiler to capture the behavior of the ring cache. Whereas HCCv1 relies on an analytical performance model to select the loops to parallelize, HCCv3 profiles loops on representative inputs. During profiling, instrumentation code emulates execution with the ring cache, resulting in an estimate of time saved by parallelization. Finally, HCCv3 uses a loop nesting graph, annotated with the profiling results, to choose the most promising loops.

5. Architecture Enhancements

Adding a ring cache to a multicore architecture enables the proactive circulation of data and signals that boost parallelization. This section describes the design of the ring cache and its constituent ring nodes. The design is guided by the following objectives:

Low-latency communication. HELIX-RC relies on fast communication between cores in a multicore processor for synchronization and for data sharing between loop iterations. Since low-latency communication is possible between physically adjacent cores in modern processors, the ring cache implements a simple unidirectional ring network.

Caching shared values. A compiler cannot easily guarantee whether and when shared data generated by a loop iteration will be consumed by other cores running subsequent iterations. Hence, the ring cache must cache shared data. Keeping shared data on local ring nodes provides quick access for the associated cores. As with data, it is also important to buffer signals in each ring node for immediate consumption.

Easy integration. The ring cache is a minimally-invasive extension to existing multicore systems, easy to adopt and integrate. It does not require modifications to the existing memory hierarchy or to cache coherence protocols.

With these objectives in mind, we now describe the internals of the ring cache and its interaction with the rest of the architecture.

5.1. Ring Cache Architecture

The ring cache architecture relies on properties of compiled code including: (i) parallelized loop iterations execute in separate threads on separate cores, arranged in a logical ring; and (ii) data shared between iterations moves between cores from current to future iterations. These properties imply that the data involved in timing-critical dependences that potentially limit overall performance are both produced and consumed in the same order as loop iterations. Furthermore, a ring network topology captures this data flow, as sketched in Figure 6. The following paragraphs describe the structure and purpose of each ring cache component.

Ring node structure. The internal structure of a per-core ring node is shown in the right half of Figure 6. Parts of this structure resemble a simple network router. Unidirectional links connect a node to its two neighbors to form the ring backbone. Bidirectional connections to the core and private L1 cache allow injection of data into and extraction of data from the ring. There are three separate sets of data links and buffers. A primary set forwards data and signals between cores. Two other sets manage infrequent traffic for integration with the rest of the memory hierarchy (see Section 5.2). Separating these three traffic types simplifies the design and avoids deadlock. Finally, signals move in lockstep with forwarded data to ensure that a shared memory location is not accessed before the data arrives.

In addition to these router-like elements, a ring node also contains structures more common to caches. A set associative *cache array* stores all data values (and their tags) received by the ring node, whether from a predecessor node or from its associated core. The line size of this cache array is kept at one machine word. While the small line is contrary to typical cache designs, it ensures there will be no false data sharing by independent values from the same line.

The final structural component of the ring node is the *signal buffer*, which stores signals until they are consumed.

Node-to-node connection. The main purpose of the ring cache is to proactively provide many-to-many core communication in a scalable and low-latency manner. In the unidirectional ring formed by the ring nodes, data propagates by *value circulation*. Once a ring node receives an (address, value) pair, either from its predecessor, or from its associated core, it stores a local copy in its cache array and propagates the same pair to its successor node. The pair eventually propagates through the entire ring (stopping after a full cycle) so that any core can consume the data value from its local ring node, as needed.

This value circulation mechanism allows the ring cache to communicate between cores faster than reactive systems (like

most coherent cache hierarchies). In a reactive system, data transfer begins once the receiver requests the shared data, which adds transfer latency to an already latency-critical code path. In contrast, a proactive scheme overlaps transfer latencies with computation to lower the receiver's perceived latency.

The ring cache prioritizes the common case, where data generated within sequential segments must propagate to all other nodes as quickly as possible. Assuming no contention over the network and single-cycle node-to-node latency, the design shown in Figure 6 allows us to bound the latency for a full trip around the ring to N clock cycles, where N is the number of cores. Each ring node prioritizes data received from the ring and stalls injection from its local core.

In order to eliminate buffering delays within the node that are not due to L1 traffic, the number of write ports in each node's cache array must match the link bandwidth between two nodes. While this may seem like an onerous design constraint for the cache array, Section 6.3 shows that just one write port is sufficient to reap more than 99% of the ideal-case benefits.

To ensure correctness under network contention, the ring cache is sometimes forced to stall all messages (data and signals) traveling along the ring. The only events that can cause contention and stalls are ring cache misses and evictions, which may then need to fetch data from a remote L1 cache. While these ring stalls are necessary to guarantee correctness, they are infrequent.

The ring cache relies on credit-based flow control [17] and is deadlock free. Each ring node has at least two buffers attached to the incoming links to guarantee forward progress. The network maintains the invariant that there is always at least one empty buffer per set of links somewhere in the ring. That is why a node only injects new data from its associated core into the ring when there is no data from a predecessor node to forward.

Node-core integration. Ring nodes are connected to their respective cores as the closest level in the cache hierarchy (Figure 6). The core's interface to the ring cache is through regular loads and stores for memory accesses in sequential segments.

As previously discussed, `wait` and `signal` instructions delineate code within a sequential segment. A thread that needs to enter a sequential segment first executes a `wait`, which only returns from the associated ring node when matching signals have been received from all other cores executing prior loop iterations. The signal buffer within the ring node enforces this. Specialized core logic detects the start of the sequential segment and routes memory operations to the ring cache.[4] Finally, executing the corresponding `signal` marks the end of the sequential segment.

The `wait` and `signal` instructions require special treatment in out-of-order cores. Since they may have system-wide side effects, these instructions must issue non-speculatively from the core's store queue and regular loads and stores cannot be reordered around them. Our implementation reuses logic from load-store queues for memory disambiguation and holds a lightweight local fence in the load queue until the `wait` returns to the senior store queue. This is not a concern for in-order cores.

[4]This feature may add one multiplexer delay to the critical delay path from the core to L1.

Benchmark	Phases	Parallel loop coverage		
		HELIX-RC	HCCv2	HCCv1
Integer benchmarks				
164.gzip	12	98.2%	42.3%	42.3%
175.vpr	28	99%	55.1%	55.1%
197.parser	19	98.7%	60.2%	60.2%
300.twolf	18	99%	62.4%	62.4%
181.mcf	19	99%	65.3%	65.3%
256.bzip2	23	99%	72.3%	72.1%
Floating point benchmarks				
183.equake	7	99%	99%	77.1%
179.art	11	99%	99%	84.1%
188.ammp	23	99%	99%	60.2%
177.mesa	8	99%	99%	64.3%

Table 1: Characteristics of parallelized benchmarks.

5.2. Memory Hierarchy Integration

The ring cache is a level within the cache hierarchy and as such must not break any consistency guarantees that the hierarchy normally provides. Consistency between the ring cache and the conventional memory hierarchy results from the following invariants: (i) shared memory can only be accessed within sequential segments through the ring cache (compiler-enforced); (ii) only a uniquely assigned *owner* node can read or write a particular shared memory location through the L1 cache on a ring cache miss (ring cache-enforced); and (iii) the cache coherence protocol preserves the order of stores to a memory location through a particular L1 cache.[5]

Sequential consistency. To preserve the semantics of a parallelized single-threaded program, memory operations on shared values require sequential consistency. The ring cache meets this requirement by leveraging the unidirectional data flow guaranteed by the compiler. Sequential consistency must be preserved when ring cache values reach lower-level caches, but the consistency model provided by conventional memory hierarchies is weaker. We resolve this difference by introducing a single serialization point per memory location, namely a unique *owner* node responsible for all interactions with the rest of the memory hierarchy. When a shared value is moved between the ring cache and L1 caches (owing to occasional ring cache load misses and evictions), only its owner node can perform the required L1 cache accesses. This solution preserves existing consistency models with minimal impact on performance.

Cache flush. Finally, to guarantee coherence between parallelized loops and serial code between loop invocations, each ring node flushes the dirty values of memory locations it owns to L1 once a parallel loop has finished execution. This is equivalent to executing a distributed fence at the end of loops. In a multiprogram scenario, signal buffers must also be flushed/restored at program context switches.

6. Evaluation

By co-designing the compiler along with the architecture, HELIX-RC more than triples the performance of parallelized code when compared to a compiler-only solution (HCCv2). This

[5]Most cache coherence protocols (including Intel, AMD, and ARM implementations) provide this minimum guarantee.

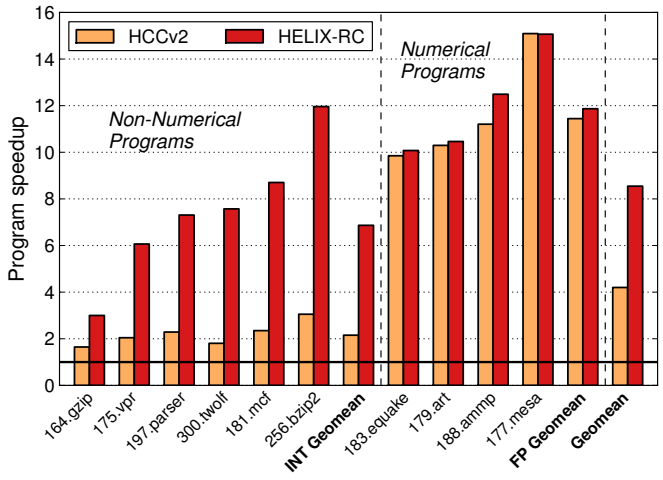


Figure 7: HELIX-RC triples the speedup obtained by HCCv2. Speedups are relative to sequential program execution.

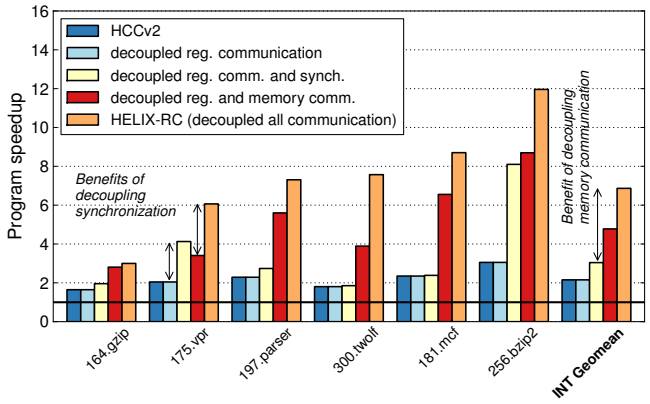


Figure 8: Breakdown of benefits of decoupling communication from computation.

section investigates HELIX-RC's performance benefits and their sensitivity to ring cache parameters. We confirm that the majority of speedups come from decoupling all types of communication and synchronization. We conclude by analyzing the remaining overheads of the execution model.

6.1. Experimental Setup

We ran experiments on two sets of architectures. The first relies on a conventional memory hierarchy to share data among the cores. The second relies on the ring cache.

Simulated conventional hardware. Unless otherwise noted, we simulate a multicore in-order x86 processor by adding multiple-core support to the XIOSim simulator. The single-core XIOSim models have been extensively validated against an Intel® Atom™ processor [19]. We use XIOSim because it is a publicly-available simulator that is able to simulate fine-grained microarchitectural events with high precision. For one of the sensitivity experiments, we also simulate out-of-order cores modeled after Intel Nehalem using the models from Zesto [21].

The simulated cache hierarchy has two levels: a per-core 32KB, 8-way associative L1 cache and a shared 8MB 16-bank L2 cache. We vary the core count from 1 to 16, but do not vary the amount of L2 cache with the number of cores, keeping it at 8MB for all configurations. Also scaling cache size would make it difficult to distinguish the benefits of parallelizing a workload from the benefits of fitting its working set into the larger cache, causing misleading results. Finally, we use DRAMSim2 [33] for cycle-accurate simulation of memory controllers and DRAM.

We extended XIOSim with a cache coherence protocol assuming an optimistic cache-to-cache latency of 10 clock cycles. This 10-cycle latency is optimistically low even compared to research prototypes of low-latency coherence [23]. In fact, it is the minimum reasonably possible with a 4×4 2D mesh network. (Running microbenchmarks in our testbed, we found that Intel Ivy Bridge is 75 cycles, Intel Sandy Bridge is 95 cycles, and Intel Nehalem is 110 cycles.) We only use this low-latency model to simulate conventional hardware, and later (Section 6.2) show that low latency alone is not enough to compensate for the lazy nature of its coherence protocol.

Simulated ring cache. We extended XIOSim to simulate the ring cache as described in Section 5. Unless otherwise noted, it has the following configuration: a 1KB 8-way associative array size, one-word data bandwidth, five-signal bandwidth, single-cycle adjacent core latency, and two cycles of core-to-ring-node injection latency to minimally impact the already delay-critical path from the core to the L1 cache. We use a simple bit mask as the hash function to distribute memory addresses to their owner nodes. To avoid triggering the cache coherence protocol, all words of a cache line have the same owner. Lastly, XIOSim simulates changes made to the core to route memory accesses either to the attached ring node or to the private L1.

Benchmarks. We use 10 out of the 15 C benchmarks from the SPEC CPU2000 suite: 4 floating point (CFP2000) and 6 integer benchmarks (CINT2000). For engineering reasons, the data dependence analysis that HCCv3 relies on [13] requires either too much memory or too much time to handle the rest. This limitation is orthogonal to the results described in this paper.

Compiler. We extended the ILDJIT compilation framework [5], version 1.1, to use LLVM 3.0 for backend machine code generation. We generated both single- and multi-threaded versions of the benchmarks. The single-threaded programs are the unmodified versions of benchmarks, optimized (O3) and generated by LLVM. This code outperforms GCC 4.8.1 by 8% on average and underperforms ICC 14.0.0 by 1.9%.[6] The multi-threaded programs were generated by HCCv3 and HCCv2 to run on ring-cache-enhanced and conventional architectures, respectively. Both compilers produce code automatically and do not require any human intervention. During compilation, they use SPEC training inputs to select the loops to parallelize.

Measuring performance. We compute speedups relative to

[6] As an aside, automatic parallelization features of ICC led to a geomean slowdown of 2.6% across SPEC CINT2000 benchmarks, suggesting ICC cannot parallelize non-numerical programs.

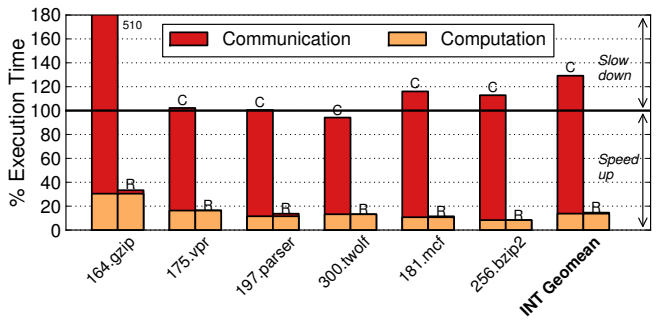


Figure 9: While code generated by HCCv3 speeds up with a ring cache (R), it slows down on conventional hardware (C).

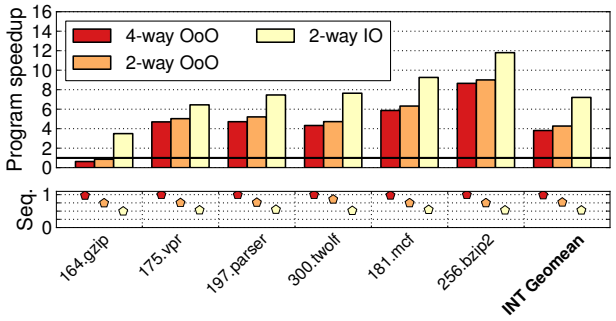


Figure 10: Speedup obtained by changing the complexity of the core from a 2-way in-order to a 4-way out-of-order.

sequential simulation. Both single- and multi-threaded runs use reference inputs. To make simulation feasible, we simulate multiple phases of 100M instructions as identified by SimPoint [14].

6.2. Speedup Analysis

In our 16-core processor evaluation system, HELIX-RC boosts the performance of sequentially-designed programs (CINT2000), assumed not to be amenable to parallelization. Figure 7 shows that HELIX-RC raises the geometric mean of speedups for these benchmarks from 2.2× for HCCv2 without ring cache to 6.85×.

HELIX-RC not only maintains the performance increases of HCCv2 (compared to HCCv1) on numerical programs (SPEC CFP2000), but also increases the geometric mean of speedups for CFP2000 benchmarks from 11.4×[7] to almost 12×.

We now turn our attention to understanding where the speedups come from.

Communication. Speedups obtained by HELIX-RC come from decoupling both synchronization and data communication from computation in loop iterations, which significantly reduces communication overhead, allows the compiler to split sequential segments into smaller blocks, and cuts down the critical path of the generated parallel code. Figure 8 compares the speedups gained by multiple combinations of decoupling synchronization, register-, and memory-based communication. As expected, fast register transfers alone do not provide much speedup since most in-register dependences can be satisfied by re-computing the shared variables involved (Section 2). Instead, most of the speedups come from decoupling communication for both synchronization and memory-carried actual dependences. To the best of our knowledge, HELIX-RC is the only solution that accelerates all three types of transfers for actual dependences.

In order to assess the impact of decoupling communication from computation for CINT2000 benchmarks, we executed the parallel code generated by HCCv3—assuming a decoupling architecture like a ring cache—on a simulated conventional system which does not decouple. The loops selected under this assumption do require frequent communication (every 24 instructions on average). Figure 9 shows that such code, running on a conventional multicore (left bars), performs no better than sequential execution (100%), even with the optimistic 10-cycle core-to-core latency. These results further stress the importance of selecting loops based on the core-to-core latency of the architecture.

Sequential segments. While more splitting offers higher TLP (more sequential segments can run in parallel), more splitting also requires more synchronization at run time. Hence, the high synchronization cost for conventional multicores discourages aggressive splitting of sequential segments.[8] In contrast, the ring cache enables aggressive splitting to maximize TLP.

To analyze the relationship between splitting and TLP, we computed the number of instructions that execute concurrently for the following two scenarios: (i) conservative splitting constrained by a contemporary multicore processor with high synchronization penalty (100 cycles) and (ii) aggressive splitting for HELIX-RC with low-latency communication (<10 cycles) provided by the ring cache. In order to compute TLP independent of both the communication overhead and core pipeline advantages, we used a simple abstracted model of a multicore system that has no communication cost and is able to execute one instruction at a time. Using the same set of loops chosen by HELIX-RC and used in Figure 7, TLP increased from 6.4 to 14.2 instructions with aggressive splitting. Moreover, the average number of instructions per sequential segment dropped from 8.5 to 3.2 instructions.

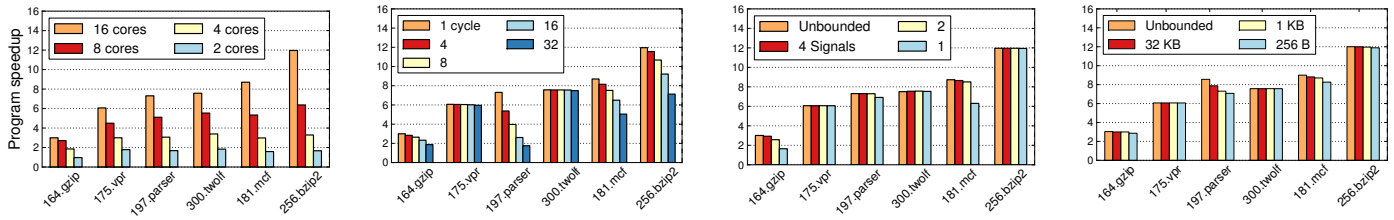
Coverage. Despite all the loop-level speedups possible via decoupling communication and aggressively splitting of sequential segments, Amdahl's law states that program coverage dictates the overall speedup of a program. Prior parallelization techniques have avoided selecting loops with small bodies because communication would slow down execution on conventional processors [6, 39]. Since HELIX-RC does not suffer from this problem, the compiler can freely select small hot loops to cover almost the entirety of the original program. Table 1 shows that HELIX-RC achieves >98% for all of the benchmarks evaluated.

6.3. Sensitivity to Architectural Parameters

Speedup results so far assumed the default configuration (in Section 6.2) for the ring cache. We now investigate the impact of different architectural parameters on speedup. In the next set of

[7] These speedups are possible even with a cache coherence latency of conventional processors (e.g., 75 cycles).

[8] This is the rationale behind DOACROSS parallelization [10].



(a) Core count. (b) Adjacent node link latency. (c) Signal bandwidth. (d) Node memory size.

Figure 11: Sensitivity to core count and ring cache parameters. Only SPEC CINT benchmarks are shown.

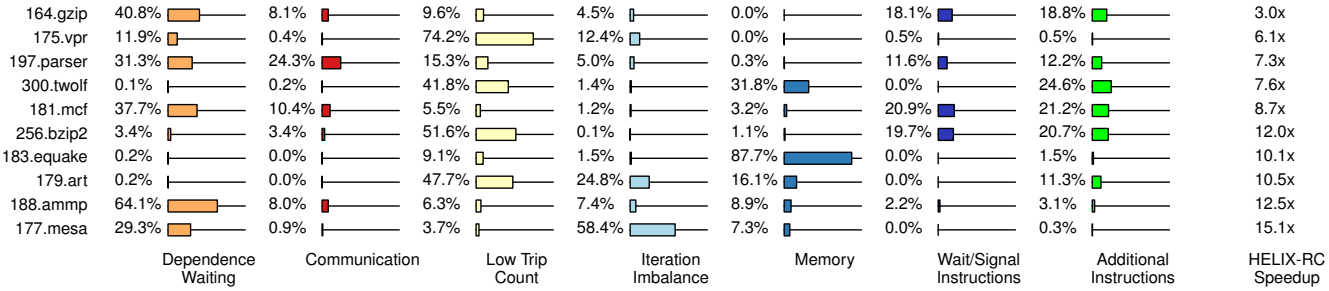


Figure 12: Breakdown of overheads that prevent achieving ideal speedup.

experiments we sweep one parameter of the ring cache at a time while keeping all others constant at the default configuration.

Core type. While HELIX-RC successfully improves TLP for in-order cores, one may ask how ILP provided by more complex cores impacts speedups. Figure 10 (upper) plots speedups for two additional core types—4-way and 2-way out-of-order (OoO) cores—in addition to the default 2-way in-order core (IO) for CINT2000 benchmarks. The lower graph plots the sequential execution time normalized to that of the 4-way OoO core. Although the OoO cores can extract more ILP for the same workloads (the 4-way OoO core is on average 1.9× faster than the default 2-way IO core), HELIX-RC speeds up most of the benchmarks except for 164.gzip. Thoroughly characterizing and accounting for this tradeoff between HELIX-RC-extracted TLP and ILP with different architectures is the subject of future work.

Core count. Figure 11a shows that HELIX-RC efficiently scales parallel performance with core count, from 2 to 16.

Link latency. Figure 11b shows the speedups obtained versus the minimum communication latency between adjacent ring nodes. As expected, HELIX-RC performance degrades for longer latencies for most of the benchmarks. It is important to note that current technologies can satisfy single-cycle adjacent core latencies, confirmed by commercial designs [46] and CACTI [24] wire models of interconnect lengths for dimensions in modern multicore processors.

Link bandwidth. A ring cache uses separate dedicated wires for data and signals to simplify design. Simulations confirm that a minimum data bandwidth of one machine word (hence, single write port) sufficiently sustains more than 99.9% of the performance obtained by a data link with unbounded bandwidth for all benchmarks. In contrast, reducing signal bandwidth can degrade performance, as shown in Figure 11c, due to synchronization stalls. However, the physical overhead of adding additional signals (up to 4) is negligible.

Memory size. Figure 11d shows the impact of memory size. The finite-size cases assume LRU replacement. Reducing cache array size within the ring node only impacts 197.parser, which has the largest ring cache working set.

6.4. Analysis of Overhead

To understand areas for improvement, we categorize every overhead cycle (preventing ideal speedup) based on a set of simulator statistics and the methodology presented by Burger et al. [4]. Figure 12 shows the results of this categorization for HELIX-RC, again implemented on a 16-core processor.

Most importantly, the small fraction of *communication* overheads suggests that HELIX-RC successfully eliminates the core-to-core latency for data transfer in most benchmarks. For several benchmarks, notably 175.vpr, 300.twolf, 256.bzip2, and 179.art, the major source of overhead is the low number of iterations per parallelized loop (*low trip count)*. While many hot loops are frequently invoked, low iteration count (ranging from 8 to 20) leads to idle cores. Other benchmarks such as 164.gzip, 197.parser, 181.mcf, and 188.ammp suffer from dependence waiting due to large sequential segments. Finally, HCCv3 must sometimes add a large number of `wait` and `signal` instructions (i.e., many sequential segments) to increase TLP, as seen for 164.gzip, 197.parser, 181.mcf, and 256.bzip2.

7. Related Work

To compare HELIX-RC to a broad set of related work, Table 2 summarizes different parallelization schemes proposed for non-numerical programs organized with respect to the types of communication decoupling implemented (register vs. memory) and the types of dependences targeted (actual vs. false). HELIX-RC

	Actual dependences	False dependences
Register	**HELIX-RC**, Multiscalar, TRIPS, T3	**HELIX-RC**, Multiscalar, TRIPS, T3
Memory	**HELIX-RC**	**HELIX-RC**, TLS-based approaches, Multiscalar, TRIPS, T3

Table 2: Only HELIX-RC decouples communication for all types of dependences.

covers the entire design space and is the only one to decouple memory accesses from computation for actual dependences.

Multiscalar register file. Multiscalar processors [38] extract both ILP and TLP from an ordinary application. While a ring cache's structure resembles a Multiscalar register file, there are fundamental differences. For the Multiscalar register file, there is a fixed and relatively small number of shared elements that must be known at compile time. Furthermore, the Multiscalar register file cannot handle memory updates by simply mapping memory to a fixed number registers without a replacement mechanism. In contrast, the ring cache does not require compile-time knowledge to handle an arbitrary number of elements shared between cores (i.e., memory locations allocated at runtime) and can readily handle register updates by deallocating a register to a memory location. In other words, HELIX-RC proposes to use a distributed cache to handle both register and memory updates.

Cache coherence protocols. The ring cache addresses an entirely different set of communication demands. Cache coherence protocols target relatively small amounts of data shared infrequently between cores. Hence, cores can communicate lazily, but the resulting communication almost always lies in the critical sequential chain. In contrast, the ring cache targets frequent and time-critical data sharing between cores.

On-chip networks. While on-chip-networks (OCNs) can take several forms, they commonly implement reactive coherence protocols [34, 37, 41, 44, 46] that do not fulfill the low-latency communication requirements of HELIX-RC. Scalar operand networks [12, 42] somewhat resemble a ring cache to enable tight coupling between known producers and consumers of specific operands, but they suffer from the same limitations as the Multiscalar register file. Hence, HELIX-RC implements a relatively simple OCN, but supported by compiler guarantees and additional logic to implement automatic forwarding.

Off-chip networks. Networks that improve bandwidth between processors have been studied extensively [36, 46]. While they work well for CMT parallelization techniques [9, 28] that require less frequent data sharing, there is less overall parallelism. Moreover, networks that target chip-to-chip communication do not meet the very different low-latency core-to-core communication demands of HELIX-RC [17]. Our results show HELIX-RC is much more sensitive to latency than to bandwidth.

Non-commodity processors. Multiscalar [38], TRIPS [35], and T3 [32] are polymorphous architectures that target parallelism at different granularities. They differ from HELIX-RC in that (i) they require a significantly larger design effort and (ii) they only decouple register-to-register communication and/or false memory dependence communication by speculating.

An iWarp system [2] implements special-purpose arrays that execute fine- and coarse-grained parallel numerical programs. However, without an efficient broadcast mechanism, iWarp's fast communication cannot reach the speedups offered by HELIX-RC.

Automatic parallelization of non-numerical programs. Several automatic methods to extract TLP have demonstrated respectable speedups on commodity multicore processors for non-numerical programs [6, 16, 27, 29, 30, 43, 49]. All of these methods transform loops into parallel threads. Decoupled software pipelining (DSWP) [27] reduces sensitivity to communication latency by restructuring a loop to create a pipeline among the extracted threads with unidirectional communication between pipeline stages. Demonstrated both on simulators and on real systems, DSWP performance is largely insensitive to latency. However, significant restructuring of the loop makes speedups difficult to predict and generated code can sometimes be slower than the original. Moreover, DSWP faces the challenges of selecting appropriate loops to parallelize and keeping the pipeline balanced at runtime. While DSWP-based approaches focus more on restructuring loops to hide communication latency [16, 27, 30], HELIX-RC proposes an architecture-compiler co-design strategy that selects the most appropriate loops for parallelization.

Combining DSWP with HELIX-RC has the potential to yield significantly better performance than either alone. DSWP cannot easily scale beyond four cores [31] without being combined with approaches that exploit parallelism among loop iterations [16] (e.g., DOALL [22]). While DSWP + DOALL can scale beyond several cores, DOALL parallelism is not easy to find in non-numerical code. Instead, DSWP + HELIX-RC presents an opportunity to parallelize a much broader set of loops.

Several TLS-based techniques [15, 18, 20, 39, 48, 49], including STAMPede, Stanford Hydra, and POSH, combine hardware-assisted thread-level speculation (TLS) with compiler optimizations to manage dependences between loop iterations executing in different threads. When the compiler identifies sources and destinations of frequent dependences, it synchronizes using `wait` and `signal` primitives; otherwise, it uses speculation. HELIX-RC, on the other hand, optimizes code assuming all dependences are actual. While we believe adding speculation may help HELIX-RC, Figure 7 shows decoupled communication already yields significant speedups without misspeculation overheads.

8. Conclusion

Decoupling communication from computation makes non-numerical programs easier to parallelize automatically by compiling loop iterations as parallel threads. While numerical programs can often be parallelized by compilation alone, non-numerical programs greatly benefit from a combined compiler-architecture approach. Our HELIX-RC prototype shows that a minimally-invasive architecture extension co-designed with a parallelizing compiler can liberate enough parallelism to make good use of 16 cores for non-numerical benchmarks commonly thought not to be parallelizable.

Acknowledgements

We thank the anonymous reviewers for their feedback on numerous manuscripts. Moreover, we would like to thank Glenn Holloway for his invaluable contributions to the HELIX project. This work was possible thanks to the sponsorship of the Royal Academy of Engineering, EPSRC and the National Science Foundation (award number IIS-0926148). Any opinions, findings, and conclusions or recommendations expressed in this material are those of the authors and do not necessarily reflect the views of our sponsors.

References

[1] Randy Allen and Ken Kennedy. *Optimizing compilers for modern architectures*. Morgan Kaufmann, 2002.

[2] Shekhar Borkar, Robert Cohn, George Cox, Sha Gleason, Thomas Gross, H. T. Kung, Monica Lam, Brian Moore, Craig Peterson, John Pieper, Linda Rankin, P. S. Tseng, Jim Sutton, John Urbanski, and Jon Webb. iWarp: An integrated solution to high-speed parallel computing. In *Supercomputing*, 1988.

[3] Matthew J. Bridges, Neil Vachharajani, Yun Zhang, Thomas Jablin, and David I. August. Revisiting the sequential programming model for multi-core. In *MICRO*, 2007.

[4] Doug Burger, James R. Goodman, and Alain Kägi. Memory bandwidth limitations of future microprocessors. In *ISCA*, 1996.

[5] Simone Campanoni, Giovanni Agosta, Stefano Crespi Reghizzi, and Andrea Di Biagio. A Highly Flexible, Parallel Virtual Machine: Design and Experience of ILDJIT. In *Software: Practice and Experience*, 2010.

[6] Simone Campanoni, Timothy M. Jones, Glenn Holloway, Vijay Janapa Reddi, Gu-Yeon Wei, and David Brooks. HELIX: Automatic Parallelization of Irregular Programs for Chip Multiprocessing. In *CGO*, 2012.

[7] Simone Campanoni, Timothy M. Jones, Glenn Holloway, Gu-Yeon Wei, and David Brooks. HELIX: Making the Extraction of Thread-Level Parallelism Mainstream. In *IEEE Micro*, 2012.

[8] Ramkrishna Chatterjee, Barbara G. Ryder, and William A. Landi. Relevant Context Inference. In *POPL*, 1999.

[9] Lynn Choi and Pen-Chung Yew. Compiler and hardware support for cache coherence in large-scale multiprocessors: Design considerations and performance study. In *ISCA*, 1996.

[10] Ron Cytron. DOACROSS: Beyond vectorization for multiprocessors. In *ICPP*, 1986.

[11] Alain Deutsch. A storeless model of aliasing and its abstractions using finite representations of right-regular equivalence relations. In *ICCL*, 1992.

[12] Paul Gratz, Changkyu Kim, Karthikeyan Sankaralingam, Heather Hanson, Premkishore Shivakumar, Stephen W. Keckler, and Doug Burger. On-Chip Interconnection Networks of the TRIPS Chip. In *IEEE Micro*, 2007.

[13] Bolei Guo, Matthew J. Bridges, Spyridon Triantafyllis, Guilherme Ottoni, Easwaran Raman, and David I. August. Practical and accurate low-level pointer analysis. In *CGO*, 2005.

[14] Greg Hamerly, Erez Perelman, and Brad Calder. How to use simpoint to pick simulation points. In *ACM SIGMETRICS Performance Evaluation Review*, 2004.

[15] Lance Hammond, Benedict A. Hubbert, Michael Siu, Manohar K. Prabhu, Michael K. Chen, and Kunle Olukotun. The Stanford Hydra CMP. In *IEEE Micro*, 2000.

[16] Jialu Huang, Arun Raman, Thomas B. Jablin, Yun Zhang, Tzu-Han Hung, and David I. August. Decoupled software pipelining creates parallelization opportunities. In *CGO*, 2010.

[17] Natalie Enright Jerger and Li-Shiuan Peh. *On-Chip Networks*. Synthesis Lectures on Computer Architecture. Morgan & Claypool, 2009.

[18] Troy A. Johnson, Rudolf Eigenmann, and T. N. Vijaykumar. Speculative thread decomposition through empirical optimization. In *PPoPP*, 2007.

[19] Svilen Kanev, Gu-Yeon Wei, and David Brooks. XIOSim: power-performance modeling of mobile x86 cores. In *ISLPED*, 2012.

[20] Wei Liu, James Tuck, Luis Ceze, Wonsun Ahn, Karin Strauss, Jose Renau, and Josep Torrellas. POSH: A TLS compiler that exploits program structure. In *PPoPP*, 2006.

[21] Gabriel H Loh, Samantika Subramaniam, and Yuejian Xie. Zesto: A cycle-level simulator for highly detailed microarchitecture exploration. In *ISPASS*, 2009.

[22] Stephen F. Lundstrom and George H. Barnes. A controllable MIMD architecture. In *Advanced computer architecture*, 1986.

[23] Milo M. K. Martin. *Token coherence*. PhD thesis, University of Wisconsin-Madison, 2003.

[24] Naveen Muralimanohar, Rajeev Balasubramonian, and Norman P. Jouppi. CACTI 6.0: A tool to model large caches. Technical Report 85, HP Laboratories, 2009.

[25] Alexandru Nicolau, Guangqiang Li, and Arun Kejariwal. Techniques for efficient placement of synchronization primitives. In *PPoPP*, 2009.

[26] Alexandru Nicolau, Guangqiang Li, Alexander V. Veidenbaum, and Arun Kejariwal. Synchronization optimizations for efficient execution on multi-cores. In *ICS*, 2009.

[27] Guilherme Ottoni, Ram Rangan, Adam Stoler, and David I. August. Automatic thread extraction with decoupled software pipelining. In *MICRO*, 2005.

[28] David K. Poulsen and Pen-Chung Yew. Data prefetching and data forwarding in shared memory multiprocessors. In *ICPP*, 1994.

[29] Arun Raman, Hanjun Kim, Thomas R. Mason, Thomas B. Jablin, and David I. August. Speculative parallelization using software multi-threaded transactions. In *ASPLOS*, 2010.

[30] Easwaran Raman, Guilherme Ottoni, Arun Raman, Matthew J. Bridges, and David I. August. Parallel-stage decoupled software pipelining. In *CGO*, 2008.

[31] Ram Rangan, Neil Vachharajani, Guilherme Ottoni, and David I. August. Performance scalability of decoupled software pipelining. In *ACM TACO*, 2008.

[32] Behnam Robatmil, Dong Li, Hadi Esmaeilzadeh, Sibi Govindan, Aaron Smith, Andrew Putnam, Doug Burger, and Stephen W. Keckler. How to Implement Effective Prediction and Forwarding for Fusable Dynamic Multicore Architectures. In *HPCA*, 2013.

[33] Paul Rosenfeld, Elliott Cooper-Balis, and Bruce Jacob. DRAMSim2: A Cycle Accurate Memory System Simulator. In *IEEE Computer Architecture Letters*, 2011.

[34] Daniel Sanchez, Richard M. Yoo, and Christos Kozyrakis. Flexible architectural support for fine-grain scheduling. In *ASPLOS*, 2010.

[35] Karthikeyan Sankaralingam, Ramadass Nagarajan, Haiming Liu, Changkyu Kim, Jaehyuk Huh, Nitya Ranganathan, Doug Burger, Stephen W. Keckler, Robert G. McDonald, and Charles R. Moore. TRIPS: A polymorphous architecture for exploiting ILP, TLP, and DLP. In *ACM TACO*, 2004.

[36] Steven L. Scott. Synchronization and Communication in the T3E Multiprocessor. In *ASPLOS*, 1996.

[37] Larry Seiler, Doug Carmean, Eric Sprangle, Tom Forsyth, Michael Abrash, Pradeep Dubey, Stephen Junkins, Adam Lake, Jeremy Sugerman, Robert Cavin, Roger Espasa, Ed Grochowski, Toni Juan, and Pat Hanrahan. Larrabee: a many-core x86 architecture for visual computing. In *ACM Transactions on Graphics*, 2008.

[38] Gurindar S. Sohi, Scott E. Breach, and T. N. Vijaykumar. Multiscalar processors. In *ISCA*, 1995.

[39] J. Gregory Steffan, Christopher Colohan, Antonia Zhai, and Todd C. Mowry. The STAMPede approach to thread-level speculation. In *ACM Transactions on Computer Systems*, 2005.

[40] J. Gregory Steffan, Christopher B. Colohan, Antonia Zhai, and Todd C. Mowry. Improving value communication for thread-level speculation. In *HPCA*, 2002.

[41] Michael Bedford Taylor, Jason Kim, Jason Miller, David Wentzlaff, Fae Ghodrat, Ben Greenwald, Henry Hoffman, Paul Johnson, Jae-Wook Lee, Walter Lee, Albert Ma, Arvind Saraf, Mark Seneski, Nathan Shnidman, Volker Strumpen, Matt Frank, Saman Amarasinghe, and Anant Ararwal. The RAW microprocessor: A computational fabric for software circuits and general-purpose programs. In *IEEE Micro*, 2002.

[42] Michael Bedford Taylor, Walter Lee, Saman P. Amarasinghe, and Anant Agarwal. Scalar Operand Networks. In *IEEE Transactions on Parallel Distributed Systems*, 2005.

[43] Georgios Tournavitis, Zheng Wang, Björn Franke, and Michael F. P. O'Boyle. Towards a holistic approach to auto-parallelization. In *PLDI*, 2009.

[44] Rob F. van der Wijngaart, Timothy G. Mattson, and Werner Haas. Light-weight communications on Intel's single-chip cloud computer processor. In *SIGOPS Operating Systems Review*, 2011.

[45] Hans Vandierendonck, Sean Rul, and Koen De Bosschere. The paralax infrastructure: Automatic parallelization with a helping hand. In *PACT*, 2010.

[46] David Wentzlaff, Patrick Griffin, Henry Hoffmann, Liewei Bao, Bruce Edwards, Carl Ramey, Matthew Mattina, Chyi-Chang Miao, John F. Brown, III, and Anant Agarwal. On-chip interconnection architecture of the tile processor. In *IEEE Micro*, 2007.

[47] Antonia Zhai, Christopher B. Colohan, J. Gregory Steffan, and Todd C. Mowry. Compiler optimization of scalar value communication between speculative threads. In *ASPLOS*, 2002.

[48] Antonia Zhai, J. Gregory Steffan, Christopher B. Colohan, and Todd C. Mowry. Compiler and hardware support for reducing the synchronization of speculative threads. In *ACM TACO*, 2008.

[49] Hongtao Zhong, Mojtaba Mehrara, Steve Lieberman, and Scott Mahlke. Uncovering hidden loop level parallelism in sequential applications. In *HPCA*, 2008.